Name(last, first): Lin, Sabrina

UCLA Computer Science Department

CS 180 **Algorithms & Complexity** **ID: 105102229**

Midterm **Total Time: 1.5 hours** **November 6, 2019**

Each problem has 20 points .

All algorithm should be described in English, bullet-by-bullet (with justification) You cannot quote any time complexity proofs we have done in class: you need to prove it yourself.

Problem 1: Describe the topological sort algorithm in a DAG. Prove its correctness. Analyze its complexity.

- The algorithm is as follows.
  - keep a set S of unvisited vertices
  - track how many incoming edges each vertex has
  - while there are unvisited vertices in S
    - add all vertices without any incoming edges, to our ordering in an arbitrary order
    - remove all such vertices x_i from S
    - update the number of incoming edges from x_i to another vertex in S

- Because the graph is directed and acyclic, there must be vertices without any incoming edges. Assume there are no vertices without incoming edges. Then there must be at least n edges, each pointing to a different vertices. However, there is a maximum of n-1 edges in an acyclic path with n vertices. Therefore, there must be a cycle. By contradiction, there must be vertices without any incoming edges.
- By removing edges and vertices, we never introduce cycles. Therefore, the set of unvisited vertices and edges is always a DAG, so it will always have vertices without any incoming edges while S is not empty.
- Assume our algorithm produces a solution such that there is an edge from x_i to x_j but x_j is put before x_i. If there is an edge from x_i to x_j, then x_j is added to our ordering before x_i because x_i will have an incoming edge at least until x_j is added to our ordering. By contradiction, our algorithm will always produce a valid solution.
- To initialize S and track the number of incoming edges of each vertex, we visit each edge and vertex a constant number of times when we add to our ordering. We also visit each edge and vertex a constant number of times, as visiting each neighbor is the same as visiting each edge. Therefore, our time complexity is $O(e+n)$.

Name(last, first): Liu, Sabrina

Problem 2: Run Merge sort on the following set of numbers. Show every step. Analyze the time complexity of merge sort on a set of **n** numbers (show every step)

9 3 6 2 4 1 5 7

- The stack looks similar to the following.
  - Sort (9,3,6,2)
    - sort (4,1,5,7)
    - merge
  - sort (9,3)
    - sort (6,2)
    - merge
      - sort (9)
      - sort (3)
      - merge → 3<9 → (3,9)
      - sort (6)
      - sort (2)
      - merge → 2<6 → (2,6)
      - merge
        - 2<3 → (2)
        - 3<6 → (2,3)
        - 6<9 → (2,3,6)
        - → (2,3,6,9)
      - sort (4,1)
      - sort (5,7)
      - merge
        - sort (4)
        - sort (1)
        - merge → 1<4 → (1,4)
        - sort (5)
        - sort (7)
        - merge → 5<7 → (5,7)
        - merge
          - 1<5 → (1)
          - 4<5 → (1,4)
          - → (1,4,5,7)
        - merge
          - 1<2 → (1)
          - 2<4 → (1,2)
          - 3<4 → (1,2,3)
          - 4<6 → (1,2,3,4)
          - 5<6 → (1,2,3,4,5)
          - 6<7 → (1,2,3,4,5,6)
          - 7<9 → (1,2,3,4,5,6,7)
          - → (1,2,3,4,5,6,7,9)

- The time complexity of merge sort is

$$T(n) = 2T(n/2) + cn$$

because at every step, we divide n in half and sort the left and right. Merging is linear because we do a constant number of comparisons. This is a recurrence relation that simplifies to

$$T(n) = cn\log n + d$$
$$= O(n\log n).$$

as follows.

$$T(n) = 2(2T(n/4) + cn) + cn$$
$$= 4T(n/4) + 3cn$$
$$\vdots$$
$$= dT(1) + cn\log n$$
$$= d + cn\log n$$
$$= O(n\log n)$$

We make this reduction a total of $\log_2 n$ times so that the term with T becomes $T(1)$, which runs in constant time, and the cn term because $cn\log n$.

Name(last, first): Lin, Sabrina

Problem 3: Suppose that you are given an algorithm as a blackbox. You cannot see how it is designed. The blackbox has the following properties: If you input any sequence of real numbers, and an integer **k**, the algorithm will answer YES or NO indicating whether there is a subset of the numbers whose sum is exactly **k**. Show how to use this blackbox to find the subset whose sum is **k**, if it exists.
You should use the blackbox O(**n**) times (where **n** is the size of the input sequence).

- we use the following algorithm.
  - keep a set S and initialize to the whole sequence
  - check S
  - if result is NO, then is no subset
  - otherwise, for each x_i in S
    - check S without x_i
    - if YES, then delete x_i
    - if NO, keep x_i in S
  - S is our subset at the end of our algorithm

5
1 3 4 6 2
3 4 6 2
3 6 2
3 2

- Assume the numbers in S at the end don't add up to k. Then it means we deleted at least one number that is in the desired subset. However, our algorithm only deletes a number if S without x_i contained the subset whose sum is k. Therefore, x_i is not in the subset. By contradiction, our algorithm will never delete a number in the desired subset.
- If the numbers in S at the end don't sum to k, then the only other situation is that we didn't delete a number that we should have deleted. However, our algorithm deletes all numbers x_i such that $S - x_i$ contains the subset, so by contradiction, our algorithm will always produce a valid solution.
- This algorithm is O(n) because we only use the blackbox n times.

Name(last, first): Liu, Sabrina

Problem 4: You have been commissioned to write a program for the next version of electronic voting software for UCLA. The input will be the number of candidates, **d**, and an array votes of size **v** holding the votes in the order they were cast where each vote is an integer from 1 to **d**. The goal is to determine if there is a candidate with a majority of the votes (more than half the votes) . You can use only a constant number of extra storage (note that **v** and **d** are not constants).Prove the correctness of your algorithm and analyze its time complexity.

1 3 5 10
1 3 0 5
2 2 2 7
1 2 3 2 3

- we use the following algorithm.
  - store a majority candidate and number of the candidate's votes with nonzero votes
  - for each vote:
    - if there is no majority candidate, add the vote to the majority candidate and set # of votes to 1
    - if there is a majority candidate
      - if the vote is for the majority candidate, increment the candidate's number of votes
      - if the vote is not for the majority candidate, decrement the candidate's number of votes
  - if there is a majority candidate, count the total votes to check for a majority
  - otherwise, there is no majority

- Assume there is a majority and candidate d_i has at least $v/2$ votes. Then if we remove a pair of votes for different candidates, we remove at most one of d_i's votes. Therefore, d_i will have at least $v/2 - 1$ votes out of $v-2$, so since $d_i > \frac{v-2}{2}$, d_i still maintains a majority. Therefore, if d_i has a majority, our algorithm will never remove d_i's majority. Our last step is to count the votes for d_i to check if d_i really has a majority. Therefore, if there is a majority, we will always find the majority.

- Assume there is no majority. Then either our algorithm produces no majority candidate, causing it to declare that there is no majority, or it produces a majority candidate, counts the votes for the candidate, and finds that the candidate does not have a majority. Therefore, if there is no majority, our algorithm will always say that there is no majority, so our algorithm will always produce a valid solution.

- We iterate through all the votes twice – once to find the majority candidate and once to check if the candidate has a majority. Each iteration performs a constant number of operations, so our algorithm is $O(v)$.

Name(last, first): Liu, Sabrina

Problem 5: Consider a sorted list of **n** integers and given integer **L**. We want to find two numbers in the list whose sum is equal to **L**. Design an efficient algorithm for solving this problem (note: an $O(n^2)$ algorithm would be trivial by considering all possible pairs). Justify your answer and analyze its time complexity.

- We use the following algorithm.
  - keep a left and right index, initialize the left to be the first integer and the right to be the last integer
  - while the left and right indexes are not the same
    - test the numbers at the two indexes
    - if they sum to L, we found the solution
    - if their sum is greater than L, then move the right index to the previous integer
    - otherwise, move the left index to the next integer
  - if we haven't found a pair, then is no solution

- If our algorithm produces a solution, it must be valid because the integers sum to L. Assume our algorithm fails to find a solution when there is a solution. Then it means we moved our indexes past the solutionpair. Assume the solution pair is x_i and x_j. Then without loss of generality, let's assume that our left index went to x_{i+1} before our right index went to x_j. Therefore, for some k, we must have had $x_i + x_{j+k} < L$. However, we know that $x_i + x_j = L$ and because the list is sorted, $x_j \leq x_{j+k}$. Therefore $x_i + x_{j+k} \geq L$. By contradiction, our algorithm will find a valid solution when there is a solution.
- If there is no solution, our algorithm will never find a pair that sums to L and will therefore correctly determine that there is no solution. Thus, our algorithm always produces the correct output.
- In the worst case, we keep comparing until the left and right indexes are the same, which is a maximum of n comparisons. All other analysis is constant, so our algorithm runs in $O(n)$ time.